Knowledge representation

lecture 4

Intro to rules and logic refresher

Tanel Tammet
TTU

Lecture overview

Why rules

Procedural and declarative rules

1st order logic refresher

Simple derivation systems for 1st order logic: resolution

RDFa and logic

Why rules

Derive new knowledge from existing knowledge

Learning: automatic creation of new rules

Inferences: apply rules to get new knowledge

Is the „derived knowledge“ really new or just an intrinsic consequence of data + rules? This is a philosophical question out of the scope of this course.

Rule examples

From simple to more and more complex ...

pub(X) => eatingplace(X)

pub(X) & openattime(X,T) => mayeatattime(X,T)

pub(X) & openattime(X,T) & country(X,'Britain') &
>(age(P),17) & popularity(X,S) =>
recommendscore('eating',X,T,P,0.8*poptorec(S))

...

Procedural & declarative

Declarative:

pub(X) => eatingplace(X)

Procedural:

```
...
d=fetchobjdata(x)
if (datahasattr_val(d,“type“,“pub“)) {
  storenewdata(x,“prop“,“eatingplace“);
}
if ....
...
```

Procedural & declarative

Procedural: database triggers

```
CREATE OR REPLACE TRIGGER tasuta_ins
BEFORE INSERT ON tasuta
REFERENCING NEW AS NEW OLD AS OLD
FOR EACH ROW
BEGIN
declare
cursor oic is select kood from isik where kood=:new.ik;
oi oic%ROWTYPE;
cursor cat is select
  category_id,
  ...
  if ca.tasuta='Y' then
          open oic;
          fetch oic into oi;
          if oic%NOTFOUND then
              insert into isik (kood) values (:new.ik);
          end if;
          close oic;
          select ticket_sequence.nextval into t_id from dual;
          insert into ticket(
           ticket_id
           ,ik
           .....
```

Declarative: rules + engine

Declarative rules like

pub(X) => eatingplace(X)

need a

- procedural reasoning engine to actually process data + rules and add new data
- concrete syntax for data and rules which the engine understands
- optional built-in procedural functions and predicates for convenience and efficiency (arithmetics, string handling etc)

Declarative vs proc handling

Pros for proc handling:

- proc handling easy to code for specific cases: no need for special syntax, engine api, etc, easy to incorporate arbitrary libraries and program snippets

Cons for procedural handling:

- recursive/iterative handling (derivation chains) hard to program for procedural cases
- proc rules handling code hard to modify and maintain
- achieving efficiency & developing optimisations is very hard work

Declarative vs proc handling

Pros for decl handling:

- recursive/iterative handling (derivation chains)
- proc rules handling and maintenance: independcy helps
- efficiency-targeted optimisations built into engine

Cons for decl handling:

- understanding and using required syntax
- understanding and using engine api-s
- hard to add your own procedural functions/preds from other languages/toolkits

Summary: engines vs your own code like SQL engines vs writing code for processing data files

Rules are code too

Another view:

rules are code in a rule-programming-language

like different, specialised Prolog's or Datalog's

How to use a rule?

In other words, what should rule derive from data?

Common ground for almost all rule systems:

(classical 1st order logic - various limitations) + various extensions

> Why classical logic? Because it allows to derive all things which generally make sense.

pub('texas')

pub(X) => eatingplace(X)

gives eatingplace('texas') and nothing more

Applying 1st order logic

There are many different - mostly equivalent axiomatizations and rule systems for logic.

However, the practical - and sufficient - way to think is simply this:

Find all possible matches with the premisses of the rule

Each match instantiates variables

Derive an instantiated consequence of the rule

Repeat for all matches and all consequences etc etc ad infinitum

Resolution method

Simple core method for practical reasoning (logical derivations)

Used as a basis for most reasoner implementations, with numerous additions / modifications / strategies / optimisations.

Just two rules:

- Generalised modus ponens (resolution rule)
- Limited instantiation (factorisation rule)

Resolution rule

Generalised modus ponens

A
A => B

C

Example:

P(b)
S(Y)
P(X) & S(X) => R(X)
--- vars instantiated X:=b, Y:=b
R(b)

Resolution rule

Example:

P(b) | G(s)
S(b)
P(X) & S(X) => R(X) | M(X)
---------- X:=b
R(b) | M(b) | G(s)

Observe that:

P(X) & S(X) => R(X) | M(X) is equivalent to

-P(X) | -S(X) | R(X) | M(X)

Resolution rule

Full rule:

A1 | A2 ... | An
-B1 | B2 ... | Bm
and unify(A1,B1) = substitution s

(A2 | ... An | B2 ... | Bm) s
cutting off unified A1 and -B1 and gluing rest together

unify(A,B) calculates the most general unifier of A and B:
it is the minimal instantiation s making As=Bs

Example:

unify(P(X,a,Y), P(Z,U,Z)) = {X:=Z, U:=a, Y:=Z }

Factorization rule

Example:

P(X) | P(a)

P(a)

Rule for gluing together two literals in the same disjunct using the minimal unifier.

A1 | A2 ... | An
and unify(A1,A2) = subst s

(A2 | ... | An) s

Prolog

Uses a highly specialised (and incomplete!) strategy of resolution for horn clauses: rules with max one literal in the consequent
(max one positive literal in a disjunct)

Derivation direction always from positive (right) side of the rule, giving instantiated antecedent as a result:

-R(a)
P(a)
P(X) => R(X)
---
derives P(a), and then finds contradiction with -P(a)

Does not derive R(a).

Answer mechanism

Prolog query ? R(X) means adding
-R(X) | Answer(X)
to facts and then searching for contradiction:

-R(X) | Answer(X)
P(a)
-P(X) | R(X)

-P(X) | Answer(X)

Answer(a)

Query: contradiction search

Observe that

Facts & Rules => Query

is a tautology iff

Facts & Rules & -Query

is a contradiction